Universidad Andrés Bello®

Facultad de Ingeniería

Análisis y explicación sobre estructuras de datos en código

Autores:

Vicente Gabriel Córdova Castillo

Hian Vicente Lart Maluenda

Profesor tutor:

Dr. Gustavo Esteban Gatica

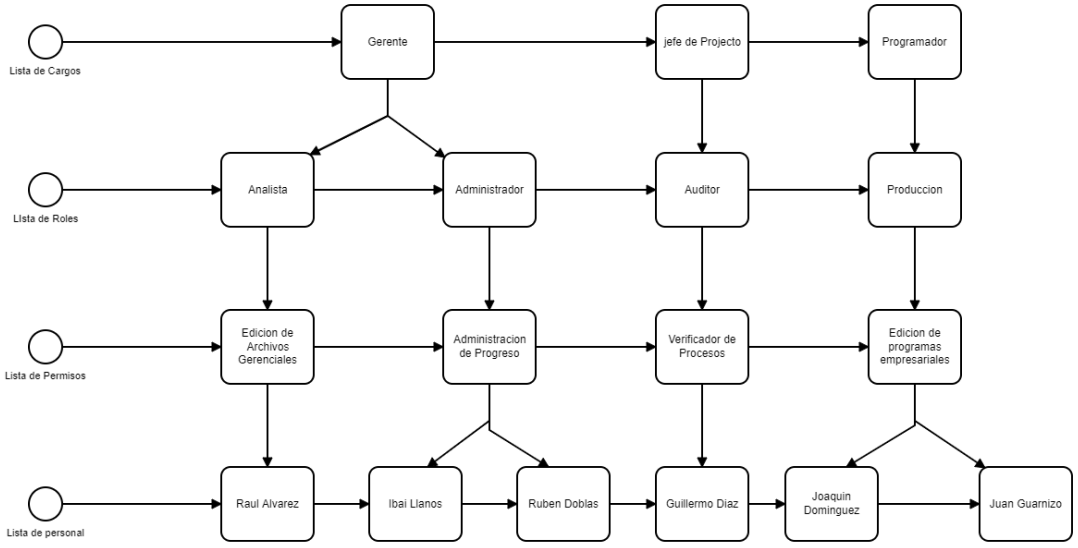
Santiago, Chile

2023

Tabla de contenido

Problema 1

Están en una empresa que desea almacenar sus cargos roles, permisos y personas en un programa en C, por lo que les solicita crear a través de “Linked List” una infraestructura que soporte los datos de la empresa. Esta tiene por finalidad poder consultar en cualquier dirección por lo que si deseo preguntar por una persona el programa debe arrojarme los permisos de ella o roles o su cargo dependiendo de la consulta; o en el caso de que pregunte por un cargo me de todos sus roles o sus permisos o las personas en ese cargo.



Con este se busca que la empresa pueda consultar por cualquier elemento de las listas y obtener los datos que están ligadas a ella por ejemplo si pregunto por el permiso de “Edición de archivos gerenciales” debería ser capaz de responderme por Raúl, Analista y gerente.

Los métodos de construcción no fueron especificados por la empresa además de las 4 Linked List y el uso de algoritmos de búsquedas presentados en la clase por lo que Uds. Pueden construirlo de la manera que quieran, pero en C.

Estructura de datos:

El grafo que observamos anteriormente presenta una estructura de nodos los cuales se encuentra entrelazados, la estructura consta de 4 listas, las cuales presentan un nodo inicial y un nodo final, los nodos de las listas se encuentran entrelazados, es decir el primer nodo de la primera lista, presenta una conexión con el ultimo nodo de la última lista.

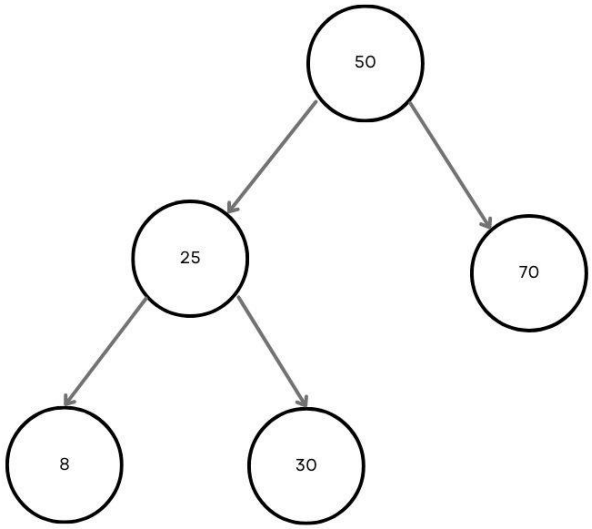
Los nodos representan puestos de la empresa, labores, funciones y nombres de los trabajadores que realizan y poseen dichos puestos.

Solución:

- La solución a nuestro problema hemos decidido inicialmente plantearla en C, posteriormente la hemos implementado en Python, para el desarrollo de nuestra solución debemos comprender la lógica que presenta el grafo presentado.
- Comenzamos creando una lista la cual puede presentar un nodo inicial o final de una máximo de 20 caracteres que estará apuntando a un nodo inicial y a un nodo final.
- Asignaremos un valor en la memoria. Podemos observar que los nodos iniciales solo presentan un nodo hacia la derecha, esto en el código está representado por "next", es decir, el recorrido de la lista avanzará hacia la derecha, por lo que siempre que nos posicionemos en el nodo inicial el único movimiento posible será "next". En los nodos interiores, es decir, ni el inicial ni el final, presentan más opciones, algunos nodos presentan conexiones con la lista inferior, es decir los nodos presentan 2 ramas inferiores, por lo que un nodo interior puede avanzar hacia adelante, atrás o hacia abajo en 2 opciones.
- La función que crea una lista enlazada se le entregará cual es el nodo inicial y cuáles son sus nodos adyacentes, en la primera lista de cargos, el cargo de "Gerente" presenta dos roles, "Analista" y "Administrador", estos 2 roles cada uno presentan permisos independientes por lo que están conectados, pero a la vez separados. Para comprender esto de mejor manera podemos decir que un cargo puede presentar más de un rol, más de un permiso, pero un permiso y un rol solo se pueden direccionar hacia un cargo individualmente.
- Un nodo inicial en la primera lista de cargos vendría siendo "Gerente" y un nodo final en esta misma lista vendría siendo "Programador", estos nodos solo podrán avanzar y bajar o retroceder y bajar, respectivamente.
- Al implementar la solución en C y Python, la búsqueda de los nodos y sus datos adyacentes, están construidos de distinta forma ya que en C tenemos un recorrido que avanza hacia la izquierda y posteriormente a la derecha, es decir, pasa sobre los mismos datos 2 veces, antes de encontrar el nodo en búsqueda, además, el muestreo se realiza de manera incorrecta ya que repite y/o omite datos. Mientras que, en Python, la búsqueda ingresamos a nuestra primera lista, y comienza a realizar la búsqueda de manera más eficiente ya que recorremos cada lista hasta encontrar el nodo solicitado, además, el muestreo de datos se presenta de manera correcta y sin repetición ni omisión de datos.

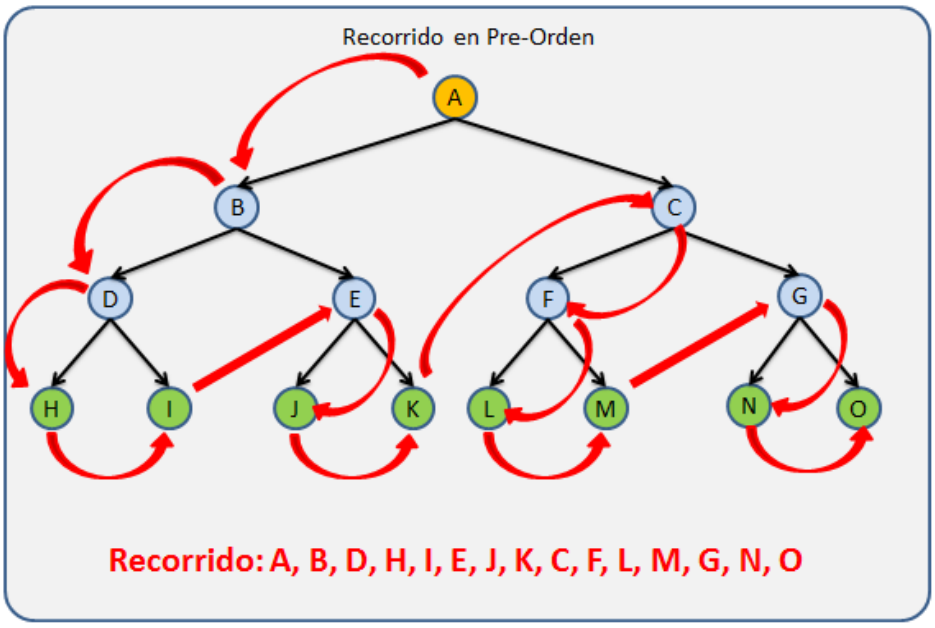
Problema 2

Se desea desarrollar un programa para la administración de un árbol ordenado con información de tipo int. Recorrer el árbol en pre, entre y post orden.



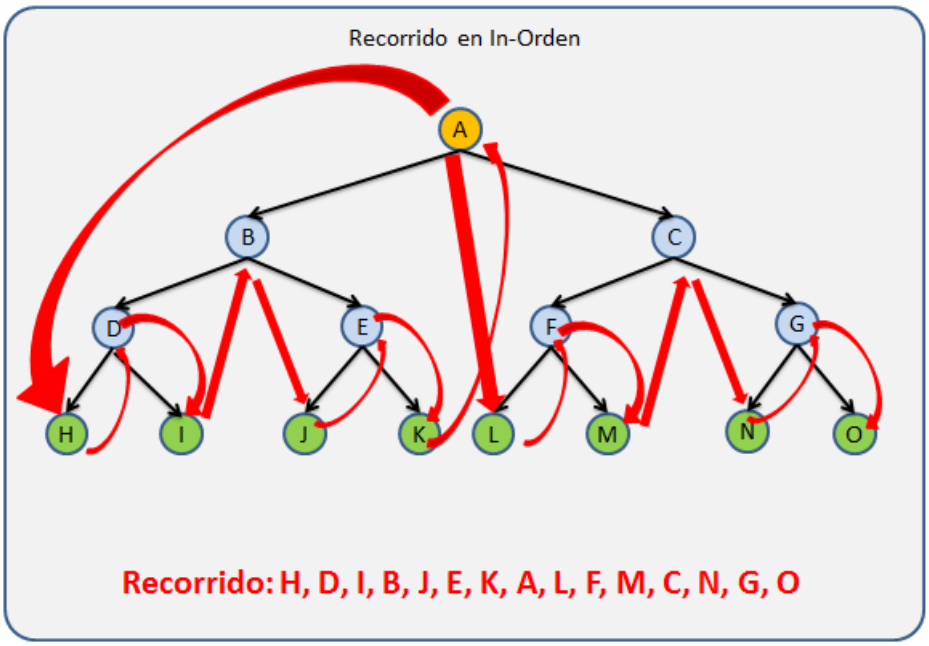
Recorrido preorden:

Se recorre desde el nodo raiz y luego va por los subarboles de la izquierda hasta la hoja del sub-árbol izquierdo, luego va por el sub arbol del nivel más alto no recorrido del lado derecho, asi sucesivamente hasta recorrer la hoja del ultimo sub-árbol derecho.



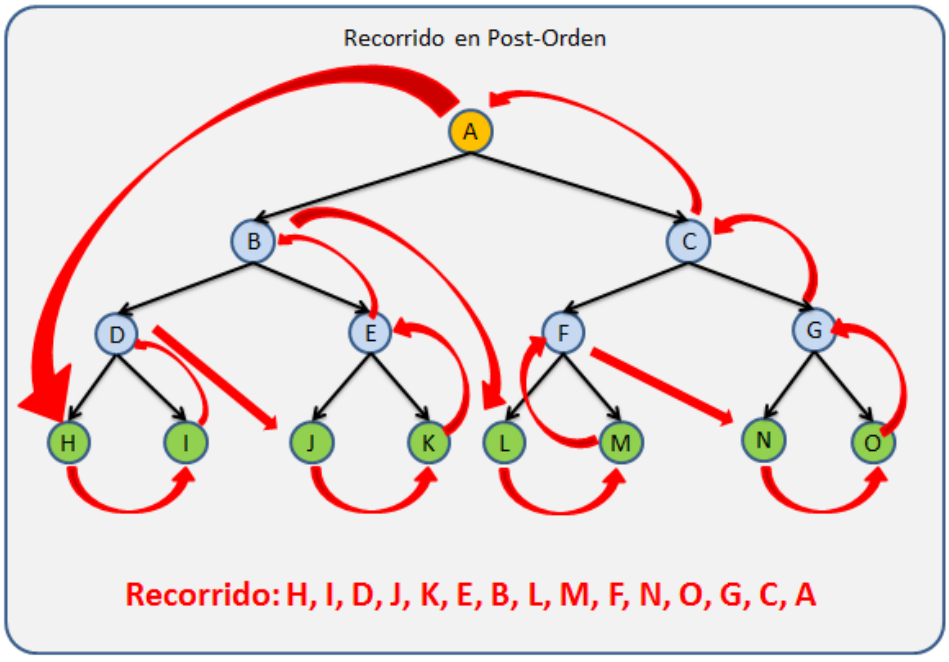
Recorrido in-orden:

Este tipo de recorrido consiste en visitar primero el subárbol izquierdo, luego la raíz y finalmente el subárbol derecho.



Recorrido post-orden:

En este recorrido se procede a visitar inicialmente el subárbol izquierdo, seguido por el subárbol derecho y, por último, se visita la raíz del árbol.



Estructura de datos

Como puede apreciarse, el nodo raíz de este árbol es el número 50. Sus nodos hijos son el 25 y el 70, lo cual los clasifica como nodos hermanos. A su vez, el nodo 25 tiene como hijos los nodos 8 y 30, los cuales también son considerados nodos hermanos. Es importante destacar que estos nodos 8 y 30 son, a su vez, nodos hojas. En cuanto a las características del árbol, se trata de un árbol binario no lleno con un peso de 5 y una altura de 3.

Solución:

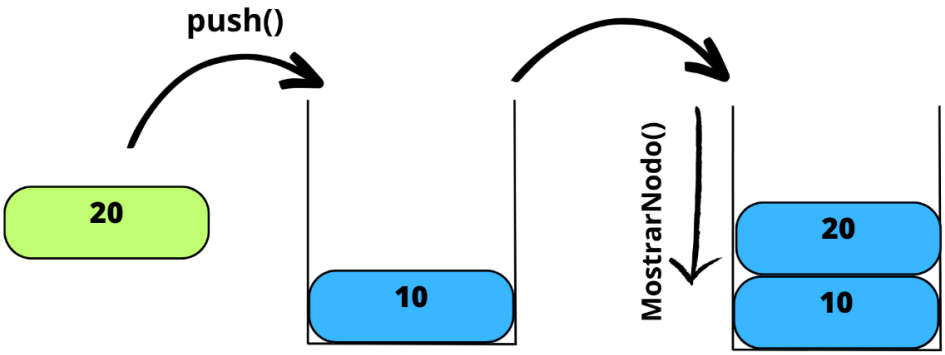
Para abordar la resolución de esta problemática, hemos desarrollado una función denominada "insertar". Dicha función solicita el número asignado al nodo a insertar. En caso de que no exista un nodo raíz previamente establecido, el primer nodo a insertar será considerado como la raíz del árbol. En caso contrario, se buscará la posición adecuada para la inserción del nuevo nodo dentro de la estructura. Además, hemos implementado funciones adicionales con el propósito de visualizar el árbol en los distintos recorridos: preorden, inorden y postorden.

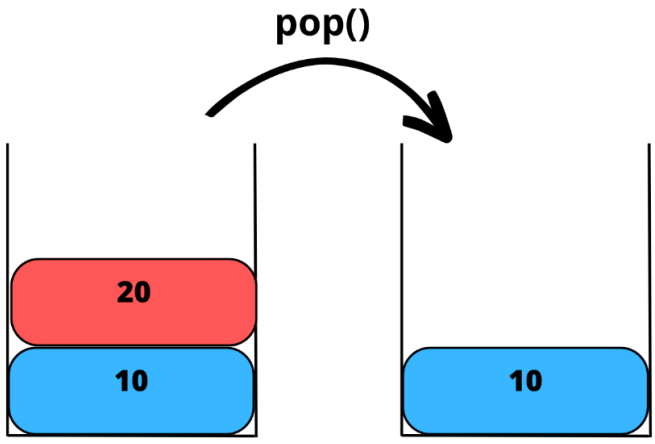
Problema 3

Se necesita implementar una estructura de datos conocida como "pila" que almacenará exclusivamente valores enteros. La pila debe contar con una función denominada "pop" que permita extraer un nodo de la pila según la solicitud del cliente.

Restricciones y requisitos:

1. La estructura de datos utilizada será una pila.
2. Los elementos almacenados en la pila serán exclusivamente números enteros.
3. La función "pop" deberá estar disponible para que el cliente pueda extraer un nodo de la pila.
4. La pila deberá ser implementada utilizando las operaciones y métodos adecuados para su correcto funcionamiento y manipulación.
5. Se requiere que la función "pop" extraiga el nodo de la pila de manera adecuada, respetando el principio "LIFO" (Last In, First Out) de las pilas.
6. En caso de que la pila esté vacía al momento de solicitar la operación "pop", se deberá manejar adecuadamente esta situación y proporcionar una respuesta o acción apropiada.





Estructura de datos

En el contexto de esta problemática, se emplea una estructura de datos conocida como pila. La pila se asemeja en ciertos aspectos a una cola, pero se distingue por su característica LIFO (Last-In, First-Out), lo cual implica que el último elemento en ser insertado en la pila será el primero en ser extraído. En otras palabras, al realizar la operación "pop", se eliminará el nodo ubicado en la cima de la pila. Cabe destacar que cada componente de la pila tiene un único sucesor y predecesor, excepto por el primer y último elemento.

Solución

Implementamos una función push() el cual va a guardar memoria para el nodo, definimos sus valores, otra función pop() para borrar el ultimo nodo de la pila y por último la última función para MostrarPila() que muestra desde el ultimo nodo hasta el primero.

Problema 4

Eres el desarrollador encargado de implementar una aplicación de gestión de productos en una tienda. Tu tarea es completar la implementación de la tabla hash para almacenar información sobre los productos disponibles.

La estructura commodity representa un producto y contiene dos campos: name (nombre del producto) y price (precio del producto). La tabla hash se utilizará para almacenar objetos de tipo commodity.

Debes completar las siguientes tareas:

Implementa la función add_node2HashTable() para agregar productos a la tabla hash. Esta función debe tomar como argumentos la tabla hash Hs_table, el nombre del producto name, la longitud del nombre key_len y el puntero al objeto commodity value. La función debe asegurarse de que no haya duplicados en la tabla y actualizar el valor del producto si ya existe. Retorna 0 si se agrega correctamente y -1 en caso de error.

Implementa la función get_value_from_hstable() para buscar un producto en la tabla hash. Esta función debe tomar como argumentos la tabla hash Hs_table, el nombre del producto name y la longitud del nombre key_len. La función debe buscar el producto correspondiente en la tabla y retornar un puntero al objeto commodity. Si el producto no se encuentra, debe retornar NULL.

Crea una función print_com_info() que tome un puntero a un objeto commodity como argumento y muestre por pantalla el nombre y el precio del producto.

Completa la función main() para probar la funcionalidad de la tabla hash. Debes crear una tabla hash utilizando la función creat_hash_table(). Luego, solicita al usuario ingresar el nombre y el precio de varios productos y agrégalos a la tabla utilizando la función add_node2HashTable(). Después, permite al usuario buscar productos en la tabla utilizando la función get_value_from_hstable() y muestra la información de los productos encontrados utilizando la función print_com_info().

Recuerda liberar la memoria ocupada por la tabla hash al finalizar la ejecución del programa utilizando la función hash_table_delete().

Consideraciones adicionales:

El código proporcionado ya incluye las estructuras y funciones necesarias para la implementación de la tabla hash, así como la función main() inicial para la interacción con el usuario.

Asegúrate de realizar las validaciones necesarias en las funciones para evitar errores y comportamientos inesperados.

Puedes definir la cantidad de productos que el usuario debe ingresar y los nombres y precios que se deben utilizar para la prueba del programa.

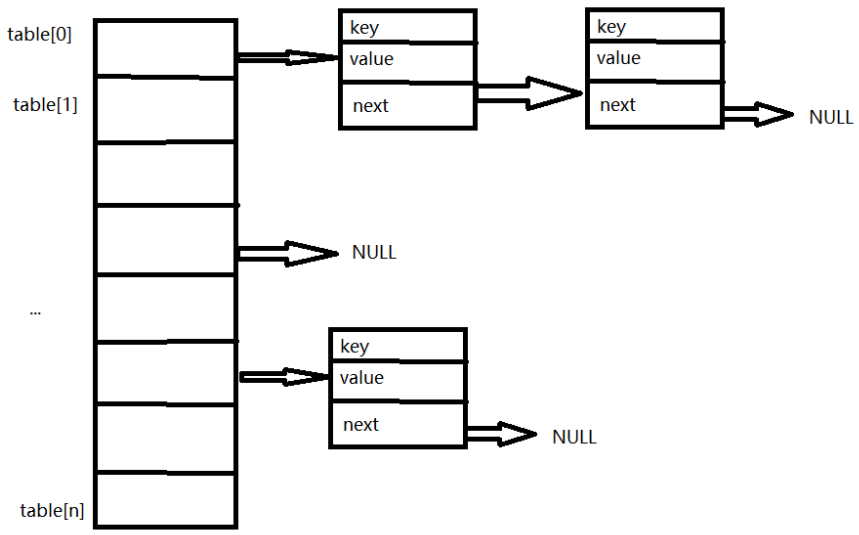
Estructura de datos

En este problema, se utiliza una tabla hash abierta para almacenar elementos basados en claves únicas. La tabla hash consiste en una estructura llamada HashNode, que contiene los atributos next, key, value y is_occupied. Estos atributos se utilizan para manejar colisiones, almacenar la clave del nodo, el valor asociado y determinar si el nodo está ocupado o no. La tabla hash en sí se implementa como un atributo llamado table, que es una lista de tamaño 1024*1024 y almacena los nodos.

La función jshash() se encarga de calcular el hash de una cadena de entrada. Comienza inicializando un valor llamado hash con una constante (1315423911). Este valor se utiliza como punto de partida para el cálculo del hash.

A continuación, la función itera sobre cada carácter de la cadena de entrada. En cada iteración, se realizan operaciones matemáticas y bit a bit en el valor de hash, combinando su valor actual con el valor del carácter actual de la cadena. Estas operaciones ayudan a mezclar los bits del hash, lo que garantiza que incluso pequeños cambios en la cadena de entrada produzcan cambios significativos en el hash resultante.

Una vez que se han recorrido todos los caracteres de la cadena, se devuelve el valor final de hash, que representa el hash calculado para la cadena de entrada.



Problema 5

La empresa anterior con la que trabajaron los contacta para actualizar su sistema de almacenamiento de trabajadores. Pero ahora desea guardar trabajadores en un sistema de árboles AVL. Para esto la empresa les solicita crear Nodos y estructurarlos en forma de árboles, cada nodo almacenará un trabajador enumerado en orden (Trabajador1, Trabajador2, Trabajador3, ETC.) y cada uno de estos tendrá un numero identificador que será creado de manera aleatoria (Valores entre el 0 - 1000). Estos números identificadores serán los valores que el árbol utilizara para ser creado.

El usuario del programa debe tener la capacidad de ingresar las cantidades de trabajadores que serán ingresado en el árbol, Ej. Si yo deseo que el programa cree 15 trabajadores el árbol deberá poseer 15 nodos uno por trabajador, tener enumerados los trabajadores del 1- 15 (Trabajador1, Trabajador2, Trabajador3, ETC.) y cada uno de estos debe poseer un numero identificador único generado de manera aleatoria (Valores entre el 0 - 1000). Además, el árbol debe tener la capacidad de equilibrarse.

- El árbol debe ser AVL y poder Equilibrarse a medida que se agregan nuevos nodos
- Se debe poder seleccionar la cantidad de Trabajadores que serán ingresados al árbol
- Es necesario tener la capacidad de consultar el árbol a través de los numero identificadores (Existe en el árbol el numero 25 o el numero 50) y retorno si no existe y si existe retornar el número del trabajadore que se le asigno. (Sugiero imprimir los valores de los números generados automáticamente para saber cuáles existen en el árbol)

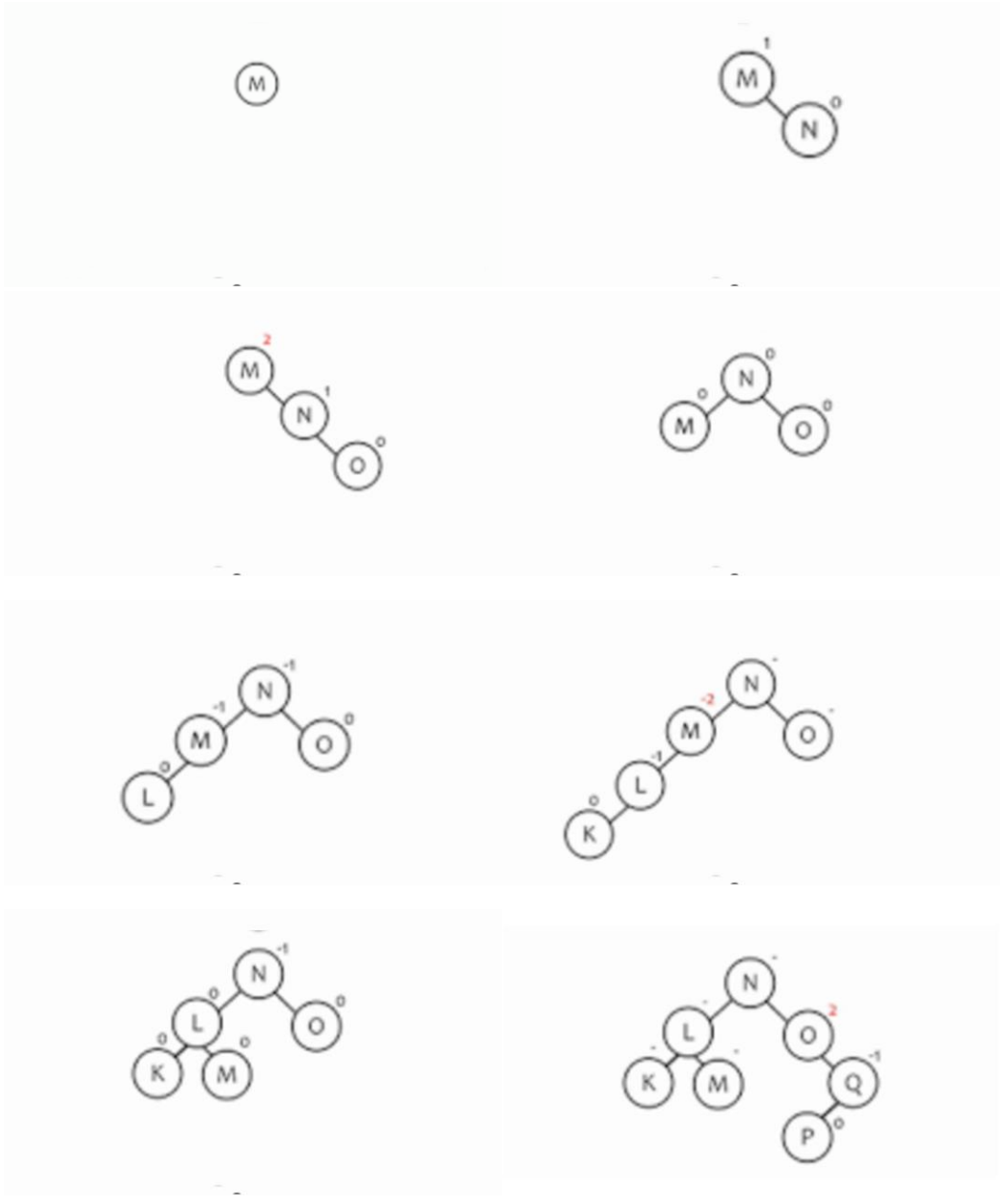
Estructura de datos

Este código implementa una estructura de datos conocida como árbol AVL (Árbol Binario de Búsqueda Balanceado) para almacenar y buscar trabajadores. El árbol AVL utiliza nodos que contienen variables como número, valor, altura, hijo izquierdo y hijo derecho.

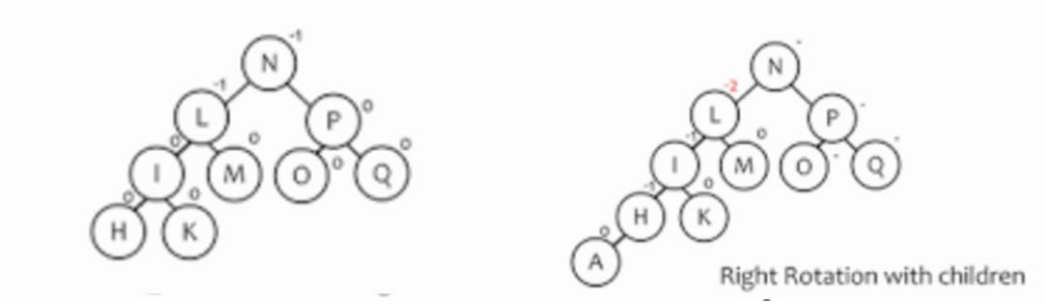
El objetivo principal de este árbol es mantener un equilibrio adecuado entre los subárboles, de modo que la diferencia de altura entre el subárbol izquierdo y el subárbol derecho sea, como máximo de 1. Para lograr esto, se emplean dos funciones clave: rotacionIzquierda() y rotacionDerecha(). Estas funciones se utilizan cuando se produce un desequilibrio en el árbol, es decir, cuando un lado tiene una altura mayor que el otro.

La función rotacionIzquierda() realiza una rotación hacia la izquierda en el árbol, donde el nodo actual se convierte en el hijo izquierdo del nodo derecho. Esto garantiza que el subárbol derecho del nodo tenga una altura menor o igual al subárbol izquierdo. Por otro lado, la función rotacionDerecha() realiza una rotación hacia la derecha, convirtiendo el nodo actual en el hijo derecho del nodo izquierdo. Esto asegura que el subárbol izquierdo del nodo tenga una altura menor o igual al subárbol derecho.

Después de una rotación, el nodo en el que se realizó la rotación puede convertirse en el nuevo nodo raíz del subárbol rotado. Esta actualización es importante, ya que afecta los punteros y las alturas de los nodos en el árbol. El nuevo nodo raíz se enlaza correctamente con el resto del árbol y garantiza que se cumplan las propiedades del árbol AVL.

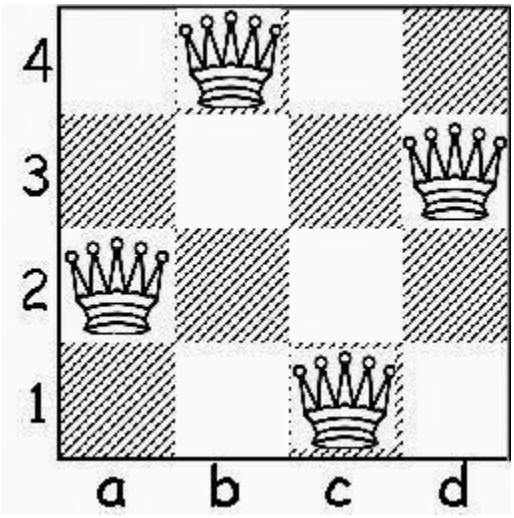



Left-Right Rotation


Right Rotation with children

Problema 6

El problema de las n reinas consiste en encontrar todas las posibles formas de colocar n reinas en un tablero de ajedrez de tamaño n x n, de manera que ninguna reina pueda amenazar a otra. Una reina puede moverse horizontalmente, verticalmente y en diagonal, por lo que no puede haber dos reinas en la misma fila, columna o diagonal.

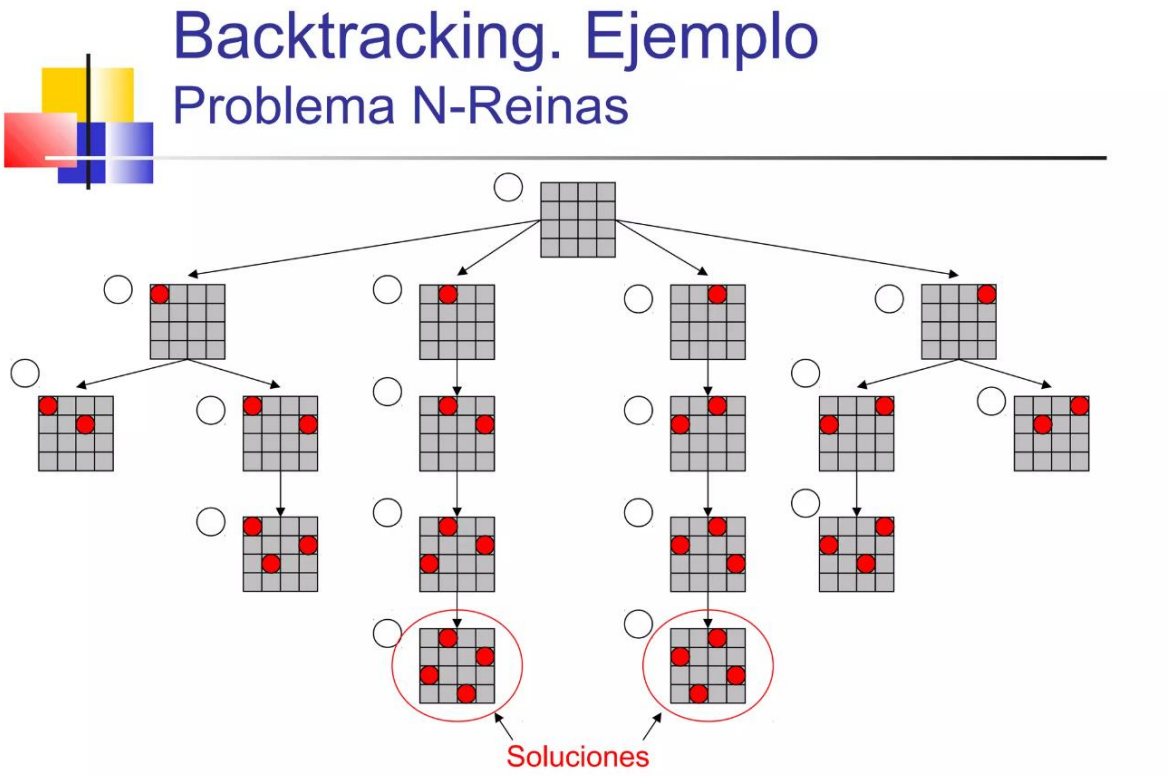


Estructura de datos

Para resolver este problema, empleamos la técnica conocida como "backtracking", la cual consiste en buscar la mejor combinación posible mediante la exploración en profundidad de un árbol. En este caso, cada nivel del árbol representa una nueva combinación, y si una combinación falla o no cumple una condición, retrocedemos al nodo (decisión) anterior y creamos un nuevo nodo que representa la siguiente combinación. En nuestro enfoque, solicitamos al usuario ingresar la cantidad de reinas que desea colocar y llenamos una lista con el valor -1, el cual indica que no se ha colocado ninguna reina. Cada posición en la lista representa una columna, y el número en esa posición indica la fila en la que se encuentra la reina.

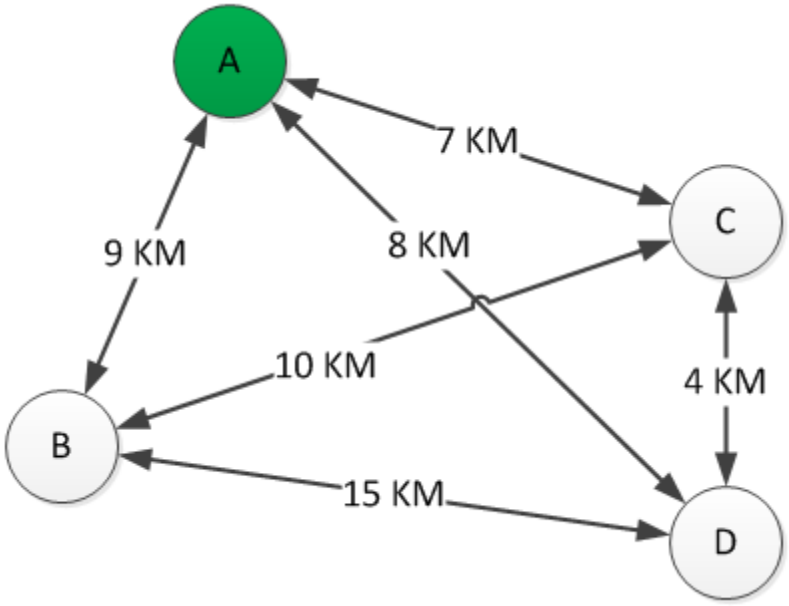
Para llevar a cabo el algoritmo de backtracking, implementamos la función "finsolution()". La primera condición de esta función verifica si ya hemos colocado todas las reinas, en cuyo caso imprime la lista con las posiciones de las reinas. La segunda condición recorre cada posición de la lista y determina en qué fila se puede colocar cada reina. Para comprobar si esto es posible, el código prueba cada una de las posibles combinaciones. Si no es posible colocar una reina, retrocede y prueba la siguiente combinación. Para verificar si una reina amenaza a otra, utilizamos la función "safe()". Esta función establece una condición que examina los valores de las posiciones anteriores para determinar si se encuentran en la misma

fila o en diagonales. Si alguna de estas dos condiciones se cumple, la función devuelve "false", lo cual indica que no se puede colocar una reina en esa posición. Si ninguna de las dos condiciones se cumple, la función devuelve "true", indicando que es posible colocar una reina en esa posición.



Problema 7

Para el problema 7 siguiente problema tenemos a una situación similar al del viajero ambulante simplemente que ahora en vez de ciudades son puertos y el que va a recorrer es un barco, tenemos un barco con cada puerto con un calado máximo, el cual es necesario para poder entrar a un puerto y poder salir, hay que pasar por todos los puertos en la menor distancia posible, la solución tiene que usar heurística para la solución.



Estructura de datos

El código proporcionado es una implementación de un algoritmo de búsqueda exhaustiva con poda (backtracking) para resolver un problema de viaje marítimo entre puertos. El objetivo es encontrar la mejor ruta para visitar todos los puertos en un barco, asegurándose de que el calado del barco (profundidad sumergida) no exceda el calado máximo permitido en cada puerto.

Para representar un puerto, se utiliza la clase puerto, que tiene tres atributos: id, que es una letra mayúscula (A, B, C, ...), calmax, que representa el calado máximo permitido en el puerto, y peso, que indica el peso adicional que se agrega al barco al visitar este puerto.

La función calcular_calado(carga_extra) se encarga de calcular el calado del barco en función del peso total del barco (incluyendo la carga adicional) y la densidad del agua. Utiliza la fórmula de Arquímedes para realizar este cálculo.

Para determinar la distancia entre dos puertos, se emplea la función calcular_distancia(puerto1, puerto2, distancias). Esta función busca en el diccionario distancias la distancia entre los puertos proporcionados. Si no hay una distancia registrada para los puertos en el diccionario, devuelve el valor "infinito".

La función principal viajero_ambulante_backtracking(puertos, distancias) realiza el algoritmo de backtracking para encontrar la mejor ruta. Aquí es donde se explora exhaustivamente todas las posibles combinaciones de puertos a visitar, asegurándose de no exceder el calado máximo en cada paso. El algoritmo utiliza una técnica de poda para evitar explorar rutas que ya son más largas que la mejor ruta actual encontrada.

Finalmente, el código lee la información de los puertos y las distancias desde un archivo de texto. Luego, se ejecuta la función viajero_ambulante_backtracking(puertos, distancias) para obtener la mejor ruta, la mejor distancia y el peso total del barco. Los resultados se imprimen en la consola, mostrando la ruta, la distancia y el tiempo que tomó el algoritmo.

Es importante mencionar que este enfoque de backtracking con poda puede ser bastante lento y no es la mejor solución para problemas grandes debido a su alta complejidad exponencial. Para conjuntos de datos más grandes, pueden utilizarse algoritmos más sofisticados, como algoritmos genéticos, heurísticas de búsqueda local o programación dinámica.

Problema 7.1

El problema 7.1 aborda una cuestión similar al problema 7; sin embargo, se emplea una heurística distinta en su enfoque.

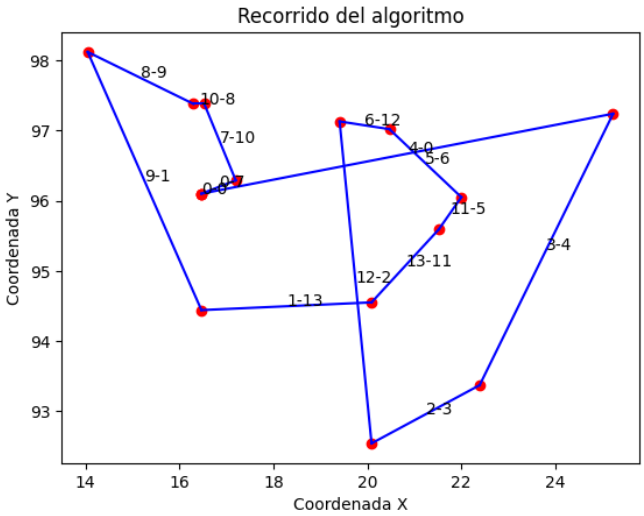
Estructura de datos

Este código implementa un algoritmo de búsqueda del vecino más cercano para resolver el problema del viajante de comercio (TSP) con la restricción de calado de un barco. La estructura de datos principal es la clase Graph, que representa un grafo que contiene nodos (puertos) y una matriz de costos para almacenar las distancias entre los nodos. Los puertos se definen en la clase Node, con atributos como las coordenadas del puerto, la demanda y el calado.

La heurística utilizada, el algoritmo de búsqueda del vecino más cercano, comienza desde un nodo inicial y en cada paso, selecciona el nodo más cercano que no haya sido visitado previamente y cumpla con la restricción de calado del barco. De esta manera, se va construyendo un recorrido que visita los puertos de manera secuencial y regresa al nodo inicial para cerrar el ciclo del TSP.

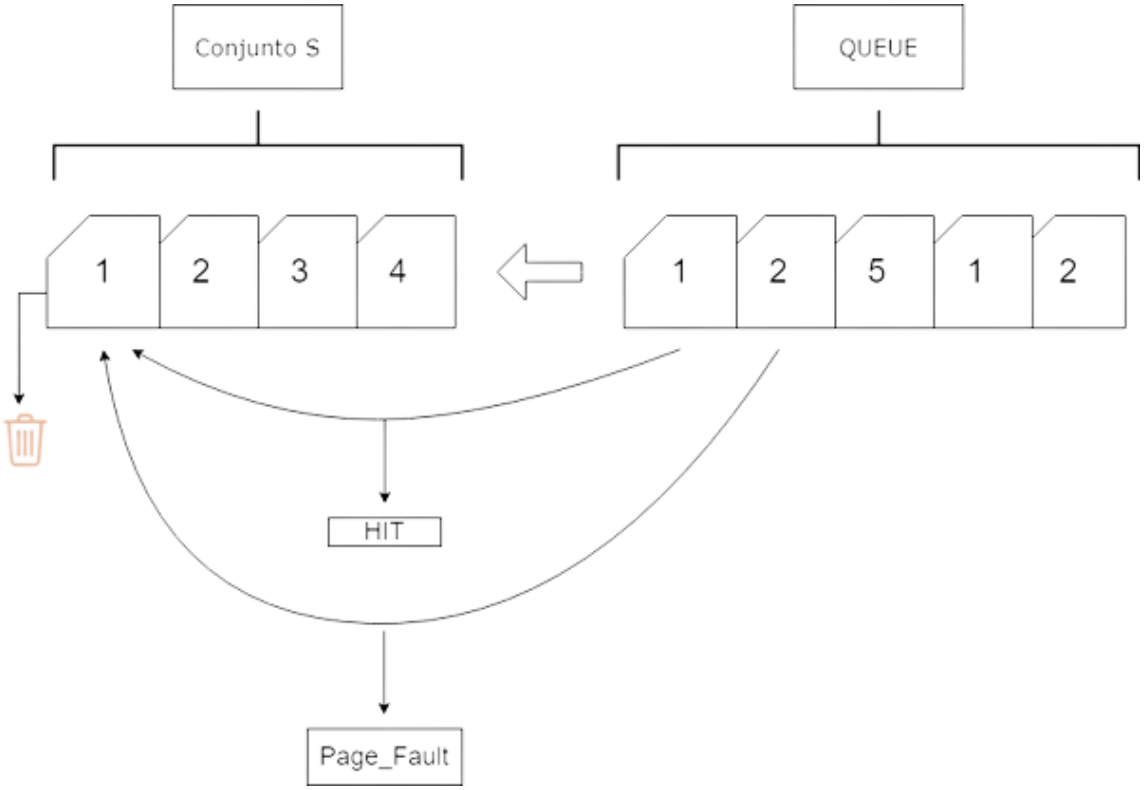
El código cuenta con funciones para cargar la información de los puertos desde un archivo de texto y construir el grafo. Luego, utiliza el algoritmo de búsqueda del vecino más cercano para encontrar el recorrido óptimo. También, se incluye una función para graficar el recorrido resultante en el grafo, lo que facilita la visualización del tour.

Es importante mencionar que el algoritmo de búsqueda del vecino más cercano no siempre proporciona la solución óptima para el TSP. Sin embargo, es una heurística efectiva para instancias pequeñas o medianas y puede dar una solución aceptable en un tiempo razonable. Para instancias más grandes, se requerirían algoritmos más avanzados para obtener soluciones cercanas a la óptima.



Problema 8

Supongamos que tienes un sistema con una memoria RAM que puede contener solo 4 páginas en total. El flujo de páginas de entrada es el siguiente: [1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5]. El código debe simular cómo se manejan las páginas en memoria utilizando el algoritmo FIFO con la limitación de 4 marcos de página.



Estructura de datos

Este código implementa el algoritmo de reemplazo de página FIFO (First-In-First-Out) que se utiliza en sistemas operativos para gestionar la memoria RAM cuando esta tiene un espacio limitado. El objetivo del algoritmo es simular cómo se manejan las páginas en memoria cuando el número de marcos de página disponibles es limitado, lo que puede causar que algunas páginas se tengan que sacar de la memoria para dar espacio a otras nuevas que llegan.

Para lograr esto, el código utiliza dos estructuras de datos principales: un conjunto (set) y una cola. El conjunto 's' se utiliza para almacenar las páginas que están actualmente presentes en la memoria RAM, lo que permite verificar rápidamente si una página está en memoria o no. La cola 'queue', por otro lado, se utiliza para mantener un registro del orden de entrada de las páginas en la memoria RAM, es decir, cuál fue la primera página que llegó y cuál fue la última.

La función 'pageFaults(incomingStream, n, frames)' es el núcleo del algoritmo. Recibe como entrada el flujo de páginas de entrada 'incomingStream', la longitud del flujo `n` y la cantidad de marcos de página disponibles en la memoria RAM 'frames'.

A medida que se recorre el flujo de páginas de entrada, el algoritmo realiza lo siguiente:

1. Si el conjunto 's' aún tiene espacio para más páginas (es decir, no se han ocupado todos los marcos de página disponibles), verifica si la página actual ya está presente en la memoria RAM. Si no lo está, la agrega al conjunto 's' y la cola 'queue', e incrementa el contador de fallos de página 'page_faults'.

2. Si el conjunto `s` está lleno (todos los marcos de página están ocupados), se verifica si la página actual ya está presente en la memoria RAM. Si no lo está, se aplica el algoritmo FIFO: se elimina la página más antigua de la cola 'queue' y del conjunto 's', y luego se agrega la nueva página a ambos. Luego, se incrementa el contador de fallos de página 'page_faults'.

El código imprime en cada paso el flujo de páginas de entrada y los marcos de página utilizados, lo que permite observar cómo se manejan las páginas en la memoria RAM a medida que se procesa el flujo.

Finalmente, el código muestra el total de fallos de página ('page_faults') y los aciertos ('hits') obtenidos durante el procesamiento del flujo de páginas. En este contexto, los aciertos son simplemente el número total de páginas que se mantuvieron en memoria y no tuvieron que ser reemplazadas debido a que ya estaban presentes.

Problema 9

Descripción:

En este problema, se te presenta una cadena de caracteres como entrada. Tu tarea es encontrar la longitud de la subcadena más larga que no contiene caracteres repetidos.

Entrada:

La entrada consistirá en una sola línea que contiene una cadena de caracteres. La cadena puede contener letras mayúsculas o letras minúsculas. La longitud de la cadena no superará los 1000 caracteres.

Estructura de datos

El código en Python busca determinar la longitud de la subcadena más larga que no contiene caracteres repetidos en una cadena dada. Esto se logra mediante el uso de dos bucles for anidados. La función longestUniqueSubsttr toma una cadena str como entrada y devuelve la longitud de la subcadena más larga sin caracteres repetidos.

Para encontrar la subcadena más larga, el primer bucle for recorre la cadena str desde el primer carácter hasta el último. Se utiliza una variable i para indicar el índice de inicio de la subcadena actual.

Dentro del primer bucle for, se inicializa una lista llamada visited, nuestra estructura de datos, con 256 elementos, que representa todos los caracteres posibles en el conjunto ASCII. Esta lista se utiliza para rastrear qué caracteres han sido visitados en la subcadena actual. Cada posición de la lista corresponde a un carácter ASCII, y los valores iniciales son False.

El segundo bucle for se inicia desde el índice actual i y recorre hasta el final de la cadena str. Esto forma una ventana deslizante sobre la cadena.

Dentro del segundo bucle for, se verifica si el carácter actual (str[j]) ya ha sido visitado en la subcadena actual, verificando el valor en la lista visited correspondiente al índice del carácter. Si el carácter ya ha sido visitado, significa que hemos encontrado una subcadena con caracteres repetidos, por lo que se rompe el bucle y se mueve al siguiente índice de inicio (i + 1) en el primer bucle for.

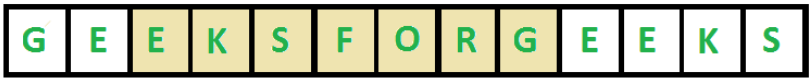
Si el carácter no ha sido visitado, se actualiza el resultado (res) para que contenga la longitud de la subcadena actual (j - i + 1) si es mayor que el valor actual de res. De esta manera, res almacenará la longitud de la subcadena más larga encontrada hasta ese momento.

El carácter actual (str[j]) se marca como visitado en la lista visited configurando el valor correspondiente a True, lo que indica que está presente en la subcadena actual.

Después de que el segundo bucle for ha terminado de iterar sobre todos los caracteres de la cadena, se restablece el valor de visited para el primer carácter de la subcadena actual (str[i]) a False. Esto se hace para prepararse para el próximo índice de inicio (i + 1) en el primer bucle for.

Finalmente, cuando el primer bucle for ha terminado de recorrer toda la cadena, el resultado (res) contendrá la longitud de la subcadena más larga sin caracteres repetidos, que se devuelve como resultado de la función.

El algoritmo aprovecha la estructura de datos de la lista visited para realizar una verificación eficiente de caracteres repetidos, lo que permite un tiempo de ejecución lineal. Esto significa que el código puede encontrar la longitud de la subcadena más larga sin caracteres repetidos de manera eficiente incluso para cadenas de longitud considerable.



Problema 10

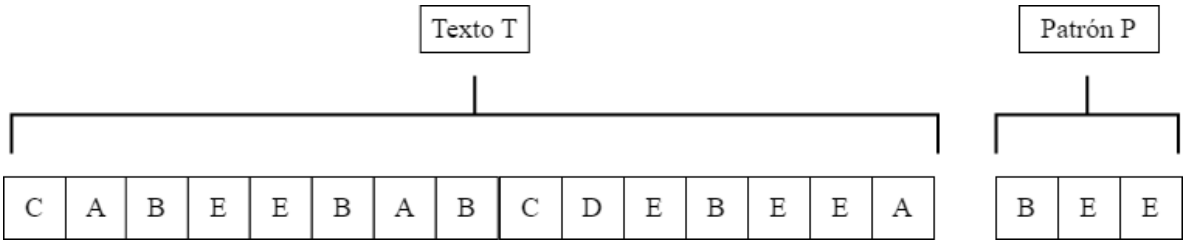
Dado un texto T y un patrón P, tu tarea es encontrar todas las posiciones en las que aparece el patrón P en el texto T.

La entrada constará de dos líneas:

La primera línea contendrá el texto T (1 <= |T| <= 10^6), una cadena de caracteres en mayúsculas y/o minúsculas.

La segunda línea contendrá el patrón P (1 <= |P| <= 10^4), una cadena de caracteres en mayúsculas y/o minúsculas.

Imprime todas las posiciones (índices) en el texto donde aparece el patrón.



Estructura de datos

El código implementa el algoritmo Z para buscar patrones en un texto dado. Este algoritmo es una técnica eficiente que ayuda a encontrar todas las ocurrencias de un patrón en un texto en tiempo lineal. La estructura de datos utilizada en el código es una lista de enteros llamada `z`, que se utiliza para almacenar los valores del arreglo Z.

El algoritmo Z funciona de la siguiente manera: Para buscar un patrón en un texto, primero concatenamos el patrón y el texto en una sola cadena. Luego, construimos el arreglo Z para esta cadena, donde cada elemento `z[i]` guarda la longitud de la subcadena más larga que comienza desde la posición `i` y que también es un prefijo de toda la cadena.

La función `getZarr(string, z)` es responsable de calcular el arreglo Z. Utiliza una ventana deslizante que mantiene un rango `[L, R]`, donde `L` es el inicio de la ventana y `R` es el final de la ventana. En cada paso, compara caracteres desde la posición `R` y actualiza los valores del arreglo Z hasta que se agote el patrón o el texto.

Luego, la función `search(text, pattern)` utiliza el arreglo Z para encontrar todas las ocurrencias del patrón en el texto. Para ello, crea una cadena concatenada con el patrón seguido de un carácter especial y el texto. Después de construir el arreglo Z, recorre este arreglo y verifica si hay ocurrencias del patrón en el texto. Si encuentra un valor `z[i]` igual a la longitud del patrón, significa que el patrón está presente en la posición `i - len(pattern) - 1` del texto original.

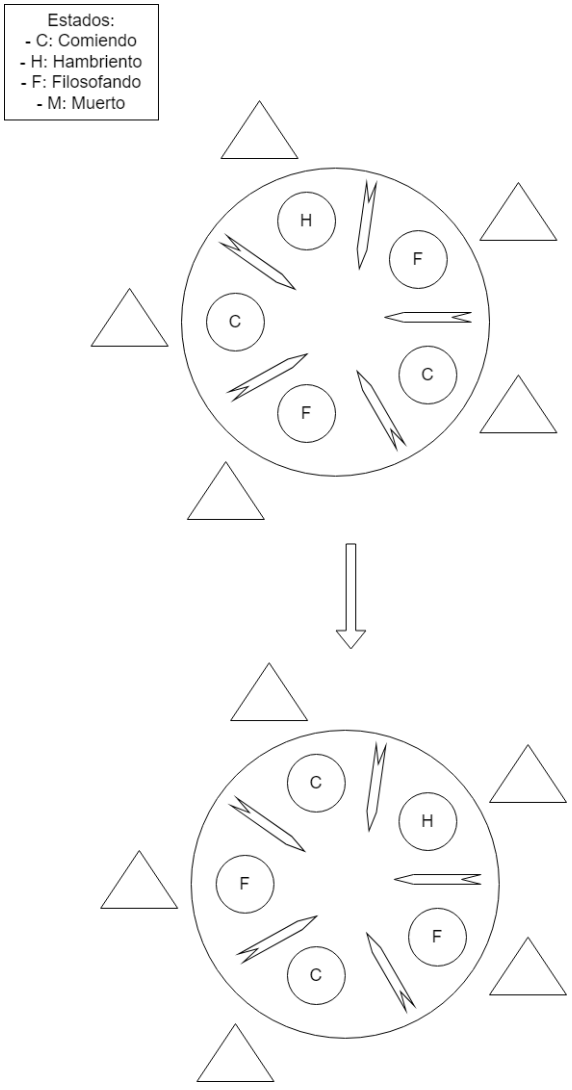
Problema 11

Cinco filósofos están sentados alrededor de una mesa y pasan su vida cenando y pensando. Cada filósofo tiene un plato de fideos y un palillo a la izquierda de su plato. Para comer los fideos son necesarios dos palillos y cada filósofo sólo puede tomar el palillo que está a su izquierda y el de su derecha.

Si cualquier filósofo toma un palillo y el otro está ocupado, se quedará esperando, con el palillo en la mano, hasta que pueda tomar el otro palillo, para luego empezar a comer. El resto de los filósofos que no está ni comiendo ni con un palillo en la mano está pensando.

El problema consiste en inventar un algoritmo que permita comer a los filósofos.

Para los fines de la práctica se agrega la condición de que si un filósofo ha intentado 10 veces obtener un palillo y no ha podido entonces este muere de inanición.



Estructura de datos

En esta simulación, la lista "estadoFilósofo" es crucial porque representa el estado actual de cada filósofo. Cada índice en la lista se refiere a un filósofo en específico, y el valor que contiene indica su estado actual. Los siguientes estados posibles son: “filosofando” mientras no está intentando comer ni esperando palillos, “hambriento” cuando intenta tomar los palillos, “comiendo” cuando ha adquirido ambos palillos, “satisfecho” cuando ha terminado de comer y está filosofando de nuevo, “muerto” cuando no puede obtener los palillos después de varios intentos.

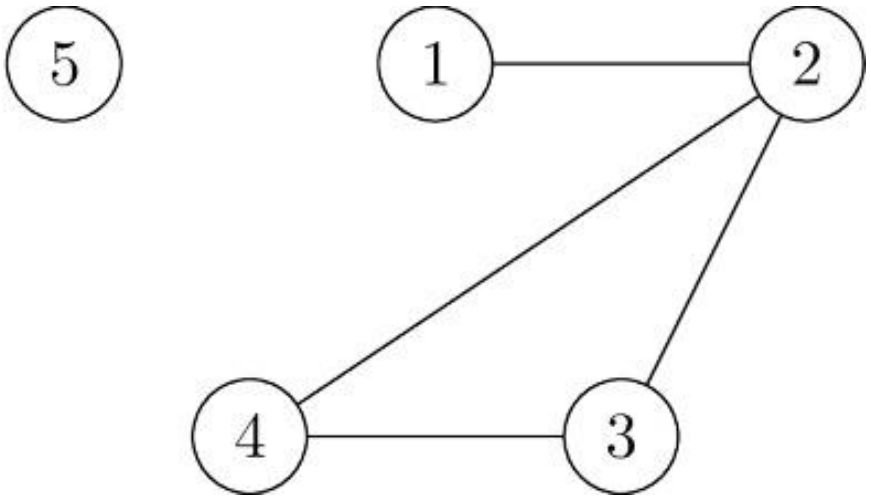
también, la lista de "candados" muestra los candados que se utilizaron para sincronizar el acceso a los palillos compartidos entre los filósofos. Cada índice de esta lista pertenece a un filósofo y contiene un candado (objeto RLock de Python) que representa el palillo a su izquierda. Además, el palillo que se encuentra a la derecha del filósofo se encuentra en el índice (id_filosofo - 1) % CANTIDAD_FILOSOFOS en la lista, La razón detrás de esta operación es que queremos obtener el índice del palillo a la derecha, asegurándonos de que el resultado esté dentro del rango válido de índices de la lista candados. Al restar 1 al id_filosofo, nos aseguramos de que lleguemos al filósofo adyacente a la derecha en la lista circular de filósofos. Los candados aseguran que solo un filósofo pueda tomar un palillo a la vez, evitando bloqueos y asegurando que los filósofos estén sincronizados.

Además, la simulación utiliza constantes y variables además de listas. Estas constantes representan la cantidad de filósofos, la cantidad máxima de intentos fallidos antes de que un filósofo muera, la cantidad de tiempo que un filósofo pasa comiendo y el tiempo total que necesita para estar satisfecho después de comer varias ráfagas. Estos parámetros se pueden ajustar para cambiar el comportamiento de la simulación y el tiempo de ejecución.

Problema 12

Dado: Un grafo simple con n≤103 vértices en el formato de lista de aristas.

Devolver: Un arreglo D[1..n] donde D[i] es la suma de los grados de los vecinos del vértice i.

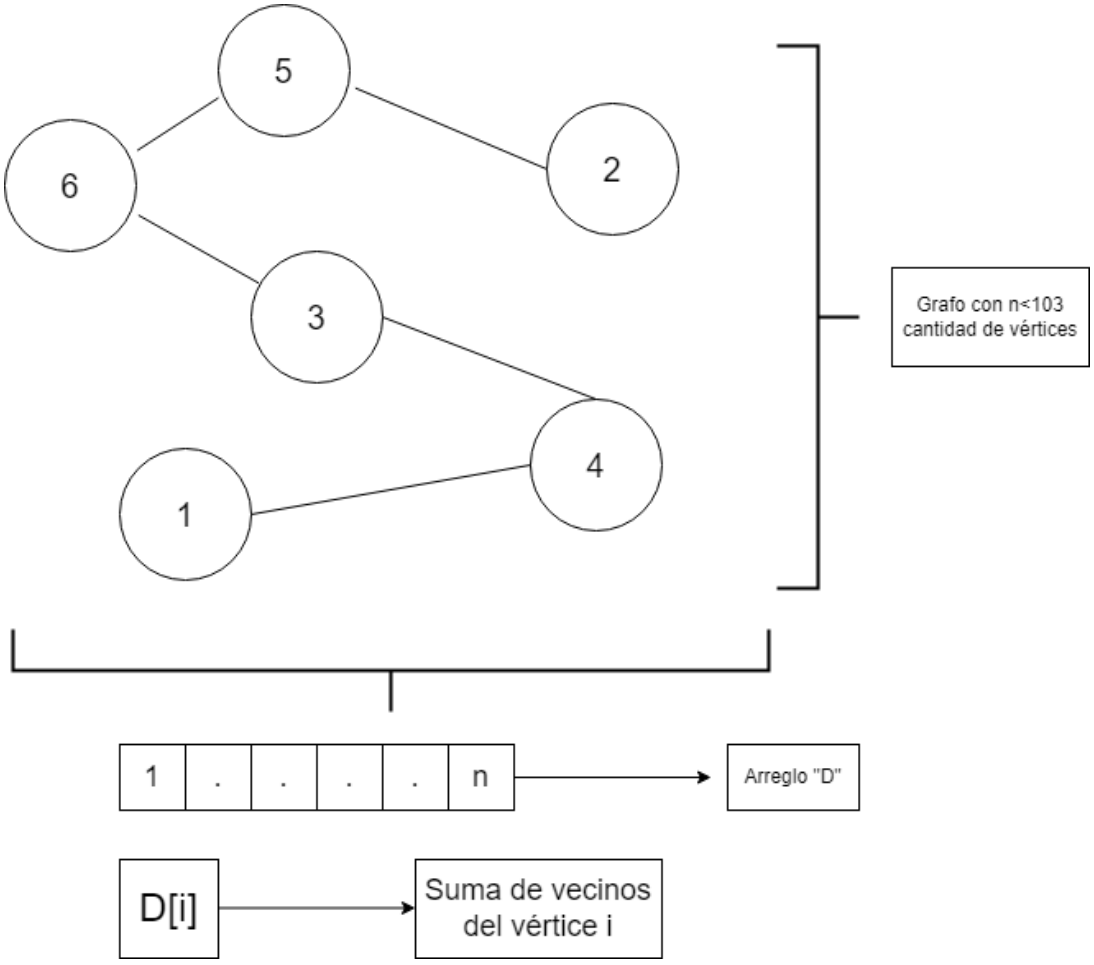


Estructura de datos

El código utiliza un bucle for que itera sobre los nodos, representados por valores enteros desde 1 hasta n. Para cada nodo, se busca en edges_list las conexiones donde el nodo es el primer elemento, pero no es el segundo. Esto se hace con una lista por comprensión que agrega los vecinos encontrados a una lista, y luego se actualiza el diccionario nodes_neighbours con el nodo como clave y la lista de vecinos como valor.

Finalmente, el código realiza otro bucle for para cada nodo en el grafo. Dentro del bucle, se imprime la suma de los grados de los vecinos para cada nodo. El grado de un nodo se calcula sumando la cantidad de vecinos que tiene, y esto se hace utilizando len(nodes_neighbours[k]), donde k representa el nodo actual.

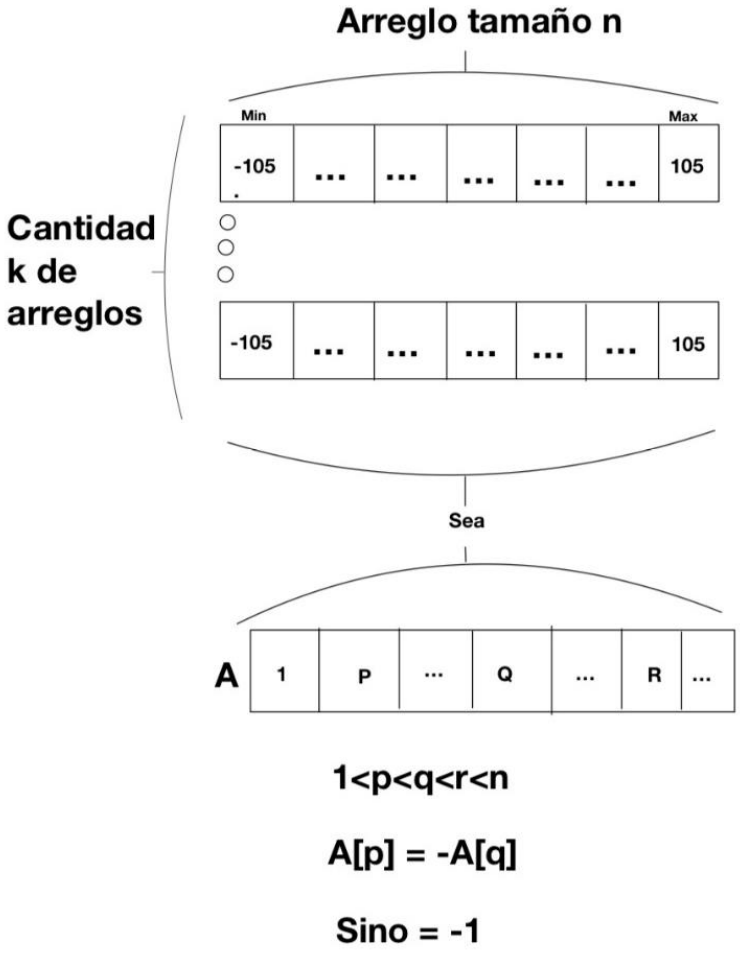
En resumen, el código utiliza listas para almacenar las conexiones entre nodos en el grafo y un diccionario para almacenar información sobre los vecinos de cada nodo. Luego, imprime la suma de los grados de los vecinos para cada nodo del grafo. Esta estructura de datos permite manipular eficientemente la información del grafo y resolver el problema relacionado con los grados de los vecinos de cada nodo.



Problema 13

Dado: Un número entero positivo k≤20, un número entero positivo n≤104, y k arreglos de tamaño n que contienen enteros en el rango de −105 a 105.

Devolver: Para cada arreglo A[1..n], mostrar dos índices diferentes 1≤p<q≤n tales que A[p]=−A[q] si existen, y "-1" en caso contrario.



Estructura de datos

La lista A es una lista de listas de enteros que contiene los datos leídos desde el archivo de entrada "rosalind_2sum.txt". Cada sublista en A representa un conjunto de números para el cual se buscará una pareja cuya suma sea igual a un valor objetivo target.

El diccionario tmp_dict se utiliza para almacenar los números del conjunto actual que se han procesado hasta el momento. La clave es el valor objetivo target - a[i], donde a[i] es el número actual del conjunto. El valor asociado a cada clave es el índice del número en la lista A.

Al leer el archivo de entrada, los valores k y n se leen en forma de enteros y se utilizan para controlar los bucles en el bloque principal del programa.

La lista A se inicializa como una lista de listas de enteros que contiene los conjuntos de números leídos desde el archivo de entrada.

La función two_sum toma como argumentos una lista de números a y un valor objetivo target (por defecto, target=0).

En la función two_sum, se utiliza el diccionario tmp_dict para almacenar los números procesados. Se recorre la lista a, y para cada número a[i], se verifica si a[i] ya está en tmp_dict. Si es así, esto significa que a[i] junto con el número que se encuentra en tmp_dict[a[i]] suman el target, y se imprime y devuelve el índice de ambos números como resultado. Si a[i] no está en tmp_dict, se agrega la clave target-a[i] al diccionario, junto con el índice i, para recordar que este número ha sido procesado.

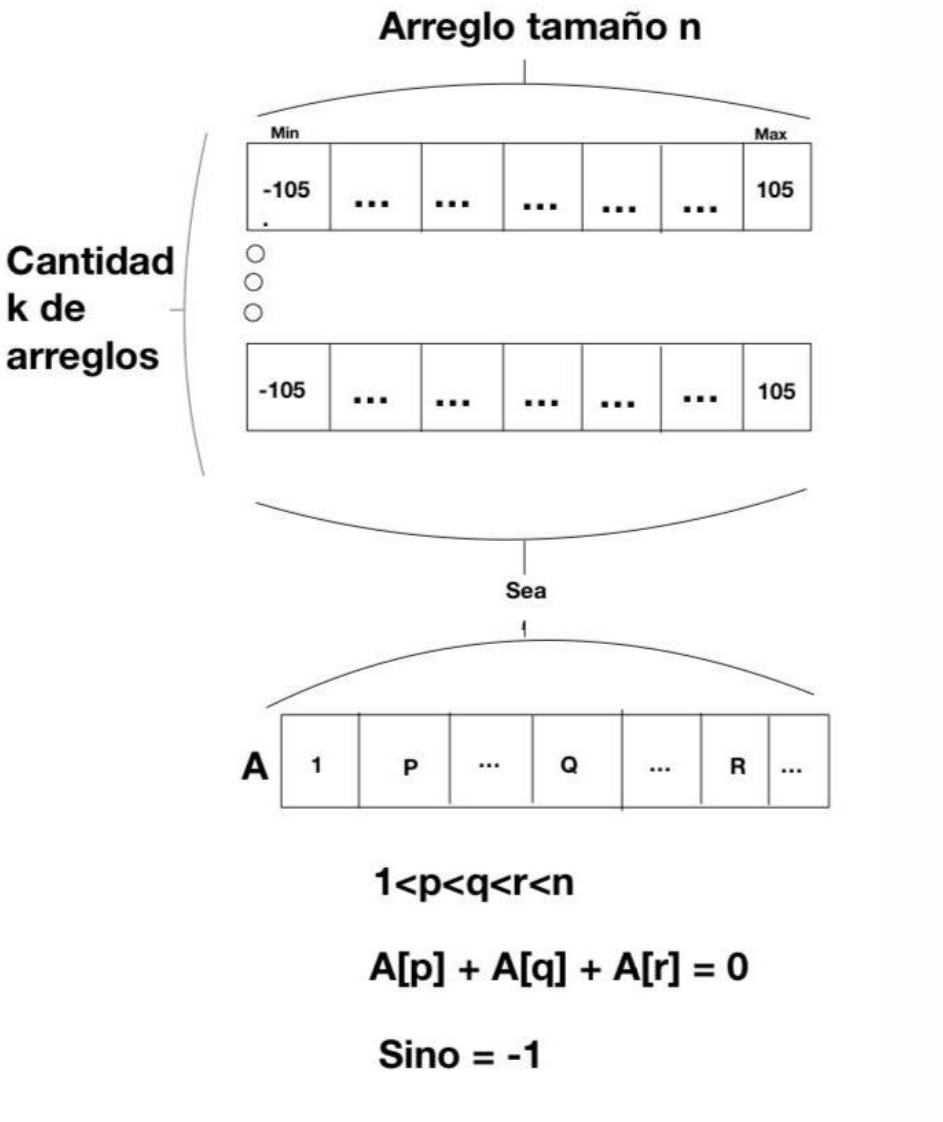
En el bloque principal del código, se leen los conjuntos de números desde el archivo de entrada y se almacenan en la lista A. Luego, para cada conjunto en A, se llama a la función two_sum pasando el conjunto actual como argumento.

En resumen, el código utiliza una lista de listas para almacenar conjuntos de números desde un archivo de entrada, y un diccionario para llevar un registro de los números procesados y sus índices. La función two_sum busca pares de números en un conjunto cuya suma sea igual a un valor objetivo utilizando el diccionario para realizar búsquedas eficientes.

Problema 14

Dado: Un número entero positivo k≤20, un número entero positivo n≤104 y k arreglos de tamaño n que contienen enteros en el rango de −105 a 105.

Devolver: Para cada arreglo A[1..n], mostrar tres índices diferentes 1≤p<q<r≤n tales que A[p]+A[q]+A[r]=0 si existen, y "-1" en caso contrario.



Estructura de datos

La función two_sum busca en un conjunto de números enteros (a) dos elementos cuya suma sea igual a un valor objetivo (target2). Para realizar esta búsqueda, utiliza un diccionario llamado tmp_dict que almacena números procesados y sus índices. Se itera sobre la lista a utilizando un bucle for, y para cada número a[i], se verifica si ya está en el diccionario. Si se encuentra un número en el diccionario que, junto con a[i], sume al target2, la función devuelve una tupla con los índices correspondientes al número encontrado y al número actual i. Si no se encuentra una suma que coincida, la función devuelve -1.

La función three_sum toma una lista de listas de números enteros (a) que representa varios conjuntos de números. Utiliza la función two_sum para buscar tres elementos en cada conjunto cuya suma sea igual a un valor objetivo (target3). Se itera sobre la lista a utilizando un bucle for, y para cada conjunto de números a[i], se llama a la función two_sum pasando el conjunto actual como argumento y target3 - a[i] como el target2. Si la función two_sum devuelve un resultado distinto de -1, la función three_sum imprime el índice i+1 y los índices calculados a partir de los resultados de la función two_sum. Luego, devuelve una tupla con los índices encontrados que representan las posiciones de los tres números cuya suma es igual al target3. Si no se encuentra un conjunto de tres números cuya suma coincida, la función imprime -1 y devuelve -1.

El código utiliza listas, diccionarios y bucles para manejar conjuntos de números, buscar combinaciones y devolver los resultados de manera adecuada. El enfoque de las funciones two_sum y three_sum permite buscar combinaciones de números que cumplan con ciertas condiciones, y el bloque principal coordina el procesamiento de los conjuntos de números leídos.

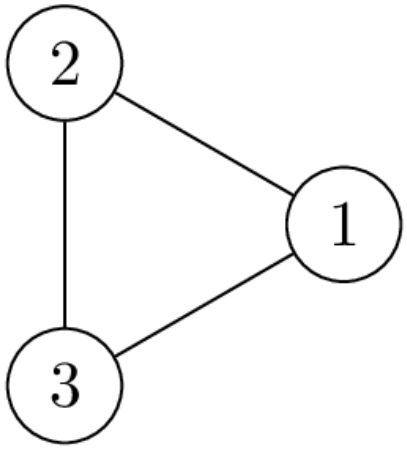
Problema 15

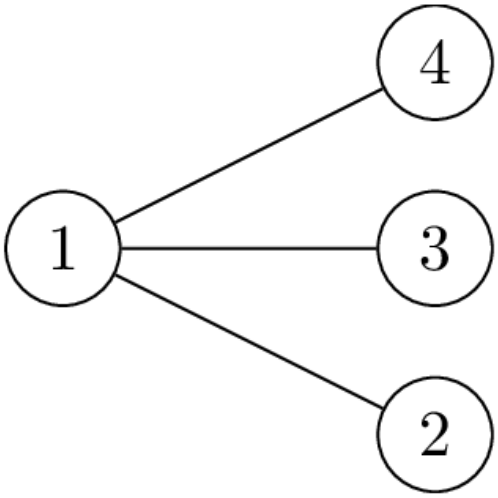
Un grafo bipartito es un grafo G=(V,E) cuyos vértices pueden ser divididos en dos conjuntos (V=V1∪V2 y V1∩V2=∅) de tal manera que no haya aristas entre vértices del mismo conjunto (por ejemplo, si u,v∈V1, entonces no hay una arista entre u y v).

Existen muchas otras formas de formular esta propiedad. Por ejemplo, un grafo no dirigido es bipartito si y solo si puede ser coloreado con solo dos colores. Otro enfoque: un grafo no dirigido es bipartito si y solo si no contiene ciclos de longitud impar.

Dado: Un número entero positivo k≤20 y k grafos simples en formato de lista de aristas, con a lo sumo 103 vértices cada uno.

Devolver: Para cada grafo, mostrar "1" si es bipartito y "-1" en caso contrario.





Estructura de datos

En primer lugar, se inicializan tres listas: graphs, vertices, y edges. Estas listas servirán para almacenar la información relacionada con los grafos que serán analizados. Luego, el código abre el archivo "rosalind_bip.txt" en modo lectura y lee el número de grafos que se encuentran en el archivo (graph_count) desde la primera línea.

A continuación, el código recorre las líneas restantes del archivo para leer la información sobre los grafos. Si una línea no está vacía, se procesa como una conexión entre dos vértices y se agrega la información a un diccionario llamado graph. Este diccionario utiliza el enfoque de lista de adyacencia, donde las claves son los vértices y los valores son listas que contienen los vértices vecinos. La idea es crear una representación del grafo a partir de las conexiones leídas.

Cuando se encuentra una línea vacía, significa que se ha terminado de leer las conexiones de un grafo. En este punto, el código lee el número de vértices (vertice) y el número de aristas (edge) para el grafo actual. Luego, se inicializa un nuevo diccionario graph para representar el siguiente grafo, y se agrega el diccionario actual a la lista graphs para almacenarlo.

Después, el código define una función llamada dfs_test_bip, que utiliza tres listas: visited, color, y el diccionario graph. La función realiza una búsqueda en profundidad (DFS) para verificar si el grafo es bipartito o no. La lista visited se utiliza para mantener un registro de los vértices visitados durante la DFS, mientras que la lista color se utiliza para asignar un color a cada vértice y distinguir los dos conjuntos (particiones) del grafo bipartito.

El bucle principal del código itera sobre cada grafo almacenado en la lista graphs. Para cada grafo, se inicializan las listas visited y color para realizar una nueva prueba de bipartición utilizando la función dfs_test_bip con el vértice 1 como punto de inicio. El resultado de la prueba de bipartición se imprime en la salida estándar. Si el grafo es bipartito, se imprime "1"; de lo contrario, se imprime "-1". La lista vertices se utiliza para obtener el número de vértices en cada grafo y asegurarse de que se realice la prueba de bipartición correctamente.

Problema 16

El formato Newick es una forma de representar árboles de manera aún más concisa que usando una lista de adyacencia, especialmente cuando se trata de árboles cuyos nodos internos no han sido etiquetados.

Primero, consideremos el caso de un árbol raíz T. Una colección de hojas v1,v2,...,vn de T son vecinas si todas están adyacentes a algún nodo interno u. El formato Newick para T se obtiene iterando el siguiente paso clave: eliminar todas las aristas {vi,u} de T y etiquetar u con (v1,v2,...,vn)u. Este proceso se repite hasta llegar a la raíz, momento en el cual un punto y coma indica el final del árbol.

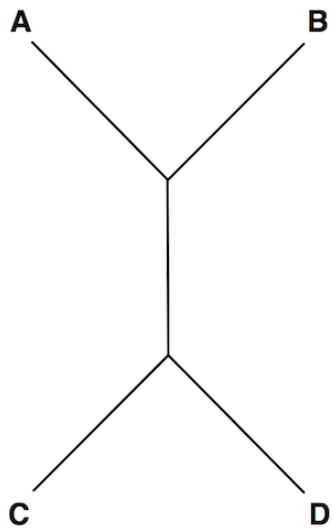
Existen varias variaciones del formato Newick. En primer lugar, si un nodo no está etiquetado en T, simplemente dejamos en blanco el espacio ocupado por el nodo. En el paso clave, podemos escribir (v1,v2,...,vn) en lugar de (v1,v2,...,vn)u si los vi están etiquetados; si ninguno de los nodos está etiquetado, podemos escribir (,,...,).

Una segunda variación del formato Newick ocurre cuando T no tiene raíz, en cuyo caso simplemente seleccionamos cualquier nodo interno para ser la raíz de T. Un caso particularmente peculiar del formato Newick surge cuando elegimos una hoja para ser la raíz.

Es importante notar que habrá una gran cantidad de formas diferentes de representar T en formato Newick; ver Figura 1.

Dado: Una colección de n árboles (n≤40) en formato Newick, donde cada árbol contiene como máximo 200 nodos; cada árbol Tk está seguido por un par de nodos xk e yk en Tk.

Devolver: Una colección de n enteros positivos, donde el k-ésimo entero representa la distancia entre xk e yk en el árbol Tk.



Estructura de datos

Se utiliza una estructura de datos basada en listas y cadenas para manejar la información y realizar los cálculos necesarios.

Primero, se define una lista llamada tree, que almacenará las líneas del archivo de entrada después de ser procesadas y formateadas. Al abrir el archivo "rosalind_nwck.txt" en modo lectura, el código recorre cada línea del archivo y elimina los caracteres de punto y coma para obtener líneas formateadas y las agrega a la lista tree. Cada línea representa un árbol en formato Newick.

A continuación, se define la función dis_tree, que tiene como argumentos una cadena que representa un árbol T, y dos nodos x e y cuya distancia se debe calcular en el árbol. La función comienza buscando las posiciones de los nodos x e y en la cadena T. Luego, se selecciona la subcadena entre las posiciones de x e y en T, y se crea otra cadena llamada bracket, que contiene únicamente los caracteres '(' (paréntesis de apertura), ')' (paréntesis de cierre), y ',' (coma) presentes en la subcadena.

A continuación, la función realiza un proceso de eliminación iterativo en la cadena bracket para eliminar los patrones '(,)', que representan un nodo con una distancia de 1 entre dos nodos. Después de esta etapa, la cadena bracket contiene únicamente los paréntesis y comas que están involucrados en la representación del camino entre los nodos x e y en el árbol.

Finalmente, se evalúan cuatro casos posibles para la cadena bracket. Si la cadena contiene solo paréntesis de apertura '(' o solo paréntesis de cierre ')', o solo comas ',', entonces significa que no hay conexión válida entre los nodos x e y en el árbol, y se devuelve un valor específico para cada caso. En el caso en que la cadena bracket contenga paréntesis de apertura y cierre en igual cantidad, esto indica que los nodos x e y son hermanos en el árbol y su distancia es igual a la cantidad de nodos que los separan en el camino, que se calcula contando los paréntesis de cierre y apertura. La distancia resultante se devuelve como resultado.

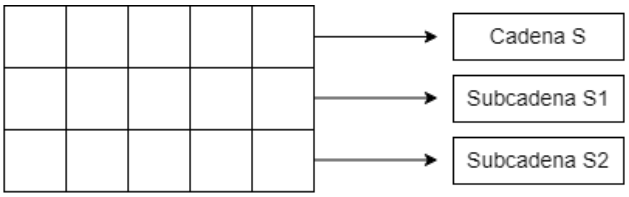
En la función main, se itera sobre la lista tree utilizando un incremento de 2 en cada iteración para obtener el árbol y los nodos x e y necesarios para calcular la distancia. Para cada árbol y par de nodos, se llama a la función dis_tree y se imprime el resultado de la distancia en la salida estándar.

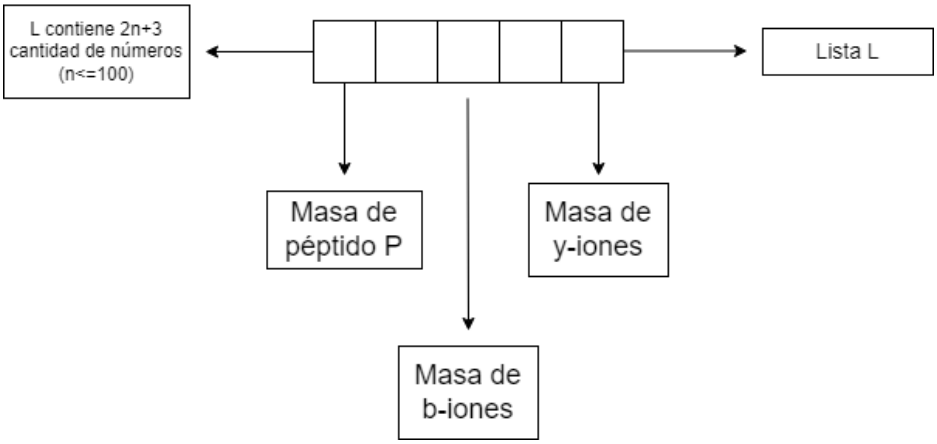
Problema 17

Supongamos que tenemos una cadena s que contiene una subcadena t, de modo que existen subcadenas no vacías s1 y s2 de s tal que s puede escribirse como s1ts2. Un prefijo t contiene toda s1 y ninguna s2; de manera similar, un sufijo t contiene toda s2 y ninguna s1.

Dado: Una lista L que contiene 2n+3 números reales positivos (n≤100). El primer número en L es la masa del péptido P (proteína), y todos los demás números representan las masas de algunos b-iones e y-iones de P (en un orden específico). Puedes asumir que, si la masa de un ion b está presente, entonces también lo está la de su ion y complementario, y viceversa.

Devolver: Una cadena de proteína t de longitud n, para la cual existen dos números reales positivos w1 y w2 tales que para cada prefijo p y sufijo s de t, cada uno de los valores w(p)+w1 y w(s)+w2 es igual a un elemento de L. (En otras palabras, existe una cadena de proteína cuyos pesos de prefijo y sufijo t corresponden a los valores de masa no padres de L). Si existen múltiples soluciones, puedes devolver cualquier una de ellas.





Estructura de datos

El programa utiliza un diccionario llamado "mass_aa" para almacenar las masas de los aminoácidos y sus correspondientes letras (códigos de una letra que representan los aminoácidos). Cada clave del diccionario es la masa del aminoácido, redondeada a 5 decimales, y el valor asociado es una lista que contiene el código de una letra del aminoácido. Algunos aminoácidos tienen más de un código de letra debido a la degeneración del código genético (por ejemplo, Ile y Leu comparten la misma masa y tienen códigos diferentes).

La función "inferring_peptide" es el núcleo del programa y se encarga de inferir el péptido a partir del espectro completo utilizando programación dinámica. A continuación, se explican los pasos principales del algoritmo:

- La función toma cuatro parámetros:
  - "n": longitud de la cadena de proteínas (peptido) objetivo.
  - "p": masa del péptido completo (parent mass).
  - "l": una lista que representa las masas de los b-iones y y-iones del espectro completo.
  - "peptide": una lista que contiene los péptidos candidatos hasta el momento.

La función utiliza recursión para construir el péptido candidato paso a paso. Comienza comprobando si la longitud del primer elemento en la lista de péptidos candidatos coincide con la longitud "n" de la cadena de proteínas objetivo. Si es así, se devuelve el péptido candidato actual, ya que representa una solución completa.

Si la longitud del péptido candidato actual no es igual a "n", la función continúa. Primero, crea una lista llamada "BYions" para almacenar todas las combinaciones posibles de aminoácidos entre las posiciones "i" y "j" de la lista "l".

Luego, para cada combinación de aminoácidos en "BYions", se actualiza la lista "l" y se agrega el nuevo aminoácido necesario a la lista de péptidos candidatos. La función se llama recursivamente con las nuevas listas "l" y "peptide".

Finalmente, la función devuelve la lista de péptidos candidatos.

En la función main, el programa carga los datos de un archivo llamado "rosalind_full.txt". El archivo contiene la masa del péptido completo en la primera línea (variable "p") y las masas de los b-iones y y-iones del espectro completo en las líneas siguientes (variable "l").

Luego, se calcula la longitud de la cadena de proteínas objetivo ("n") a partir de la lista "l". Se llama a la función "inferring_peptide" para obtener todas las soluciones candidatas, y se imprime el número total de soluciones y una solución aleatoria.

El programa utiliza un diccionario para representar las masas de los aminoácidos, una función recursiva para inferir el péptido a partir del espectro completo.

Problema 18

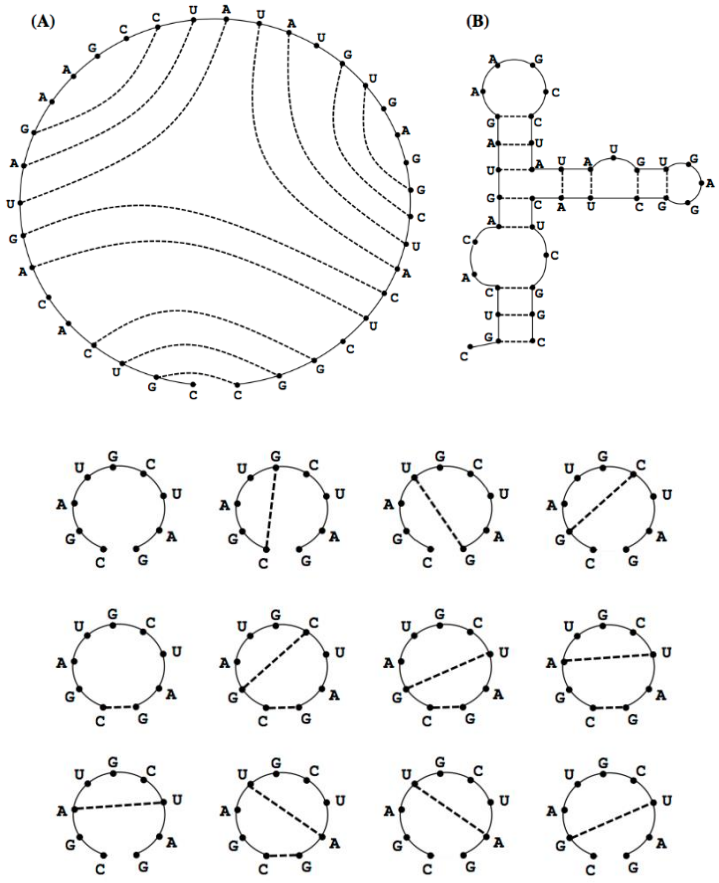
Dado un ARN string s, aumentaremos el grafo de enlace de s agregando aristas de pares de bases conectando todas las ocurrencias de 'U' con todas las ocurrencias de 'G' para representar posibles pares de bases wobble.

Decimos que un emparejamiento en el grafo de enlace para s es válido si no se cruzan (para evitar pseudonudos) y tiene la propiedad de que una arista de par de bases en el emparejamiento no puede conectar los símbolos sj y sk a menos que $k \geq j + 4$ (para evitar que nucleótidos cercanos se emparejen).

Ver Figura para un ejemplo de un emparejamiento válido si permitimos pares de bases wobble. En este problema, deseamos contar todos los posibles emparejamientos válidos en un grafo de enlace dado; ver Figura 2 para todos los posibles emparejamientos válidos en un grafo de enlace pequeño, asumiendo que permitimos el emparejamiento de bases wobble.

Dado: Un ARN string s (de longitud como máximo de 200 pb).

Resultado: El número total de emparejamientos válidos distintos de aristas de pares de bases en el grafo de enlace de s. Suponemos que se permite el emparejamiento de bases wobble.



Estructura de datos

El código proporcionado contiene una función llamada ‘get_bonding_graph’ que está diseñada para calcular el número de formas en las que una cadena de ARN puede formar puentes de Wobble. Estos puentes de Wobble son interacciones especiales entre las bases nitrogenadas en el ARN, permitiendo un cierto grado de flexibilidad en el emparejamiento de bases. El proceso de cálculo se basa en encontrar combinaciones específicas de pares de bases que pueden formar estos puentes.

La función utiliza la técnica de recursión, dividiendo gradualmente la cadena de ARN en subcadenas más pequeñas para analizarlas. También hace uso de la memorización para evitar recalcular los resultados de subcadenas que ya han sido procesadas previamente. Esto se logra mediante el uso de un diccionario denominado ‘wobble_memo’, donde las claves son las subcadenas de ARN y los valores son los resultados calculados anteriormente para esas subcadenas.

Cuando la función recibe una cadena de ARN, primero verifica si la longitud de la cadena es menor o igual a 3. Si es así, devuelve 1, ya que no hay suficientes bases para formar un puente de Wobble. Si la cadena es más larga, verifica si ya se ha calculado el resultado para esa subcadena en ‘wobble_memo’. Si es el caso, simplemente retorna el valor almacenado para evitar el cálculo repetido.

Luego, la función explora todas las combinaciones posibles de pares de bases que pueden formar puentes de Wobble. Utiliza un bucle para iterar sobre diferentes posiciones en la cadena de ARN y verifica si el par de bases en esas posiciones cumple con las condiciones específicas para formar un puente de Wobble. Si se encuentra una combinación válida, la función realiza dos llamadas recursivas utilizando subcadenas que excluyen el par de bases elegido, calculando así el número de formas en que esa combinación particular puede contribuir al total de puentes de Wobble para toda la cadena.

Una vez que se ha calculado el número de formas en que todas las combinaciones válidas pueden contribuir al total, la función almacena ese resultado en ‘wobble_memo’ para su uso futuro y lo devuelve como el resultado final para la cadena de ARN original.

En el bloque principal del programa, se lee la cadena de ARN desde un archivo llamado "rosalind_rnas.txt". Luego, se crea un diccionario vacío ‘wobble_memo’ para almacenar los resultados de las subcadenas procesadas. La función ‘get_bonding_graph’ se llama con la cadena de ARN y el diccionario de memorización, y el resultado se imprime en la consola.

El resultado final del cálculo del número de formas en que la cadena de ARN puede formar puentes de Wobble se muestra en la primera ejecución de la función.

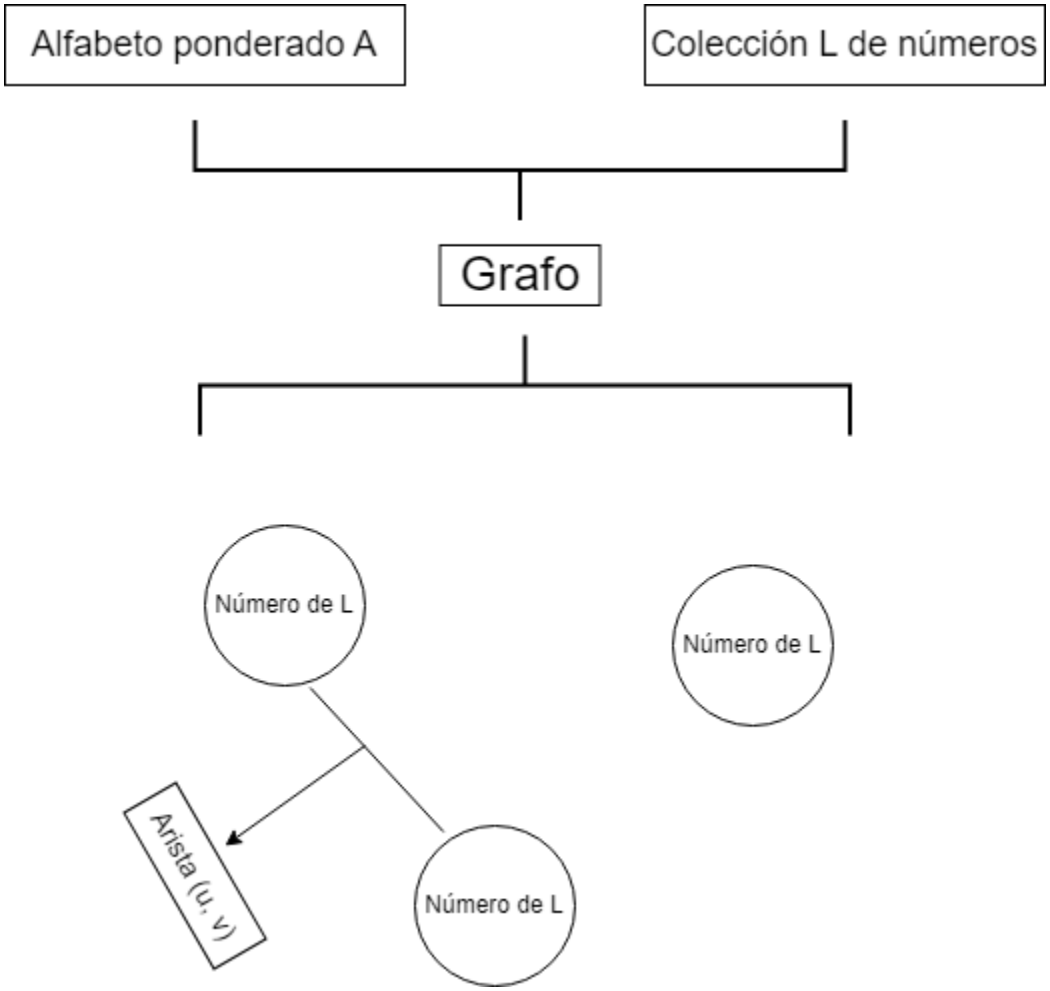
Problema 19

Para un alfabeto ponderado A y una colección L de números reales positivos, el grafo de espectro de L es un grafo dirigido construido de la siguiente manera. Primero, crea un nodo para cada número real en L. Luego, conecta un par de nodos con una arista dirigida (u,v) si v>u y v−u es igual al peso de un solo símbolo en A. Luego, podemos etiquetar la arista con este símbolo.

En este problema, decimos que un weighted string s=s1s2⋯sn coincide con L si existe una secuencia creciente de números reales positivos (w1, w2, …, wn+1) en L tal que w(s1)=w2−w1, w(s2)=w3−w2, ..., y w(sn)=wn+1−wn.

Dado: Una lista L (de longitud como máximo 100) que contiene números reales positivos.

Resultado: La cadena proteica más larga que coincide con el grafo de espectro de L (si existen múltiples soluciones, puedes mostrar cualquiera de ellas). Consulta la tabla de masas monoisotópicas.



Estructura de datos

Se definen dos diccionarios: ‘mass_aa’ y ‘mass_aa_decimal2’. El diccionario ‘mass_aa’ mapea las masas de los aminoácidos a sus correspondientes códigos de una letra. Cada clave del diccionario representa una masa de aminoácido, mientras que el valor asociado es un código de una letra que representa al aminoácido. Vale mencionar que algunos aminoácidos están representados por códigos de más de una letra debido a la degeneración del código genético. El segundo diccionario, `mass_aa_decimal2`, es una versión redondeada a 2 decimales del primero, que permite mapear las masas del espectro (valores decimales con precisión de 2 decimales) a los códigos de una letra de los aminoácidos correspondientes.

Posteriormente, el código define tres funciones: ‘_get_graph(l)’, ‘dfs_find_all_path(graph, start, end, path=[])’ y ‘_get_peptides(l, graph, edges)’. La función ‘_get_graph(l)’ se encarga de construir un grafo y un conjunto de bordes (edges) a partir de la lista de masas `l`. El grafo representa las conexiones entre las masas en el espectro, mientras que los bordes indican los aminoácidos correspondientes a las conexiones entre dos masas.

La función ‘dfs_find_all_path(graph, start, end, path=[])’ realiza una búsqueda en profundidad (DFS) en el grafo para encontrar todos los caminos posibles entre dos nodos ‘start’ y ‘end’. Utiliza recursión para explorar las diferentes combinaciones y devuelve una lista que contiene todas las rutas posibles encontradas.

Finalmente, la función ‘_get_peptides(l, graph, edges)’ utiliza el grafo y los bordes previamente construidos para inferir todos los péptidos posibles a partir del espectro de masas `l`. Itera a través de las masas en el espectro, busca todos los caminos posibles entre cada par de masas y luego traduce esos caminos en secuencias de aminoácidos utilizando el diccionario `mass_aa_decimal2`. Los péptidos inferidos se almacenan en una lista.

el programa carga los datos desde un archivo llamado "rosalind_sgra.txt", que contiene una lista de masas (valores decimales) representando el espectro. Luego, utiliza las funciones previamente definidas para obtener todos los péptidos inferidos a partir del espectro. El péptido más largo se selecciona utilizando la función ‘max(peptides, key=len, default="")’ con la longitud de la cadena de caracteres como criterio de selección.

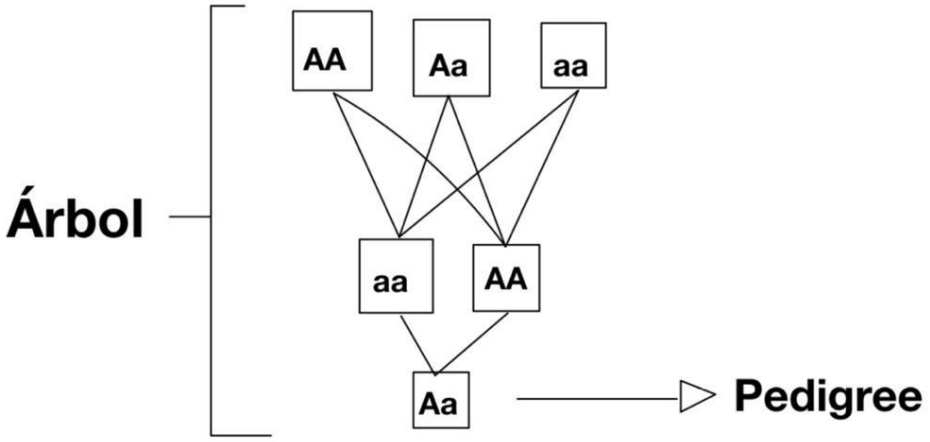
Problema 20

Un árbol binario con raíz se puede utilizar para modelar el pedigree de un individuo. En este caso, en lugar de que el tiempo progrese desde la raíz hasta las hojas, el árbol se ve al revés con el tiempo progresando desde los ancestros de un individuo (en las hojas) hasta el individuo (en la raíz).

Un ejemplo de un pedigree para un único factor en el que solo se proporcionan los genotipos de los ancestros se muestra en la Figura 1.

Dado: Un árbol binario con raíz T en formato Newick que codifica el pedigree de un individuo para un factor mendeliano cuyos alelos son A (dominante) y a (recesivo).

Resultado: Tres números entre 0 y 1, correspondientes a las respectivas probabilidades de que el individuo en la raíz de T presente los genotipos "AA", "Aa" y "aa"



Estructura de datos

La estructura de datos principal es el diccionario prob_dict, donde se almacenan las probabilidades de transmisión de los genotipos de los padres a su descendencia. El diccionario utiliza tuplas como claves para representar las combinaciones de genotipos de los padres, y los valores asociados son tuplas que contienen las probabilidades de que la descendencia tenga el genotipo "AA", "Aa" y "aa" respectivamente.

La función cp(ancestor1, ancestor2) es esencial para calcular las probabilidades de los genotipos de la descendencia. Toma dos diccionarios, ancestor1 y ancestor2, que representan los posibles genotipos de dos ancestros en el árbol genealógico. Mediante una combinación de bucles y acceso al diccionario prob_dict, esta función calcula las probabilidades de heredar cada uno de los genotipos posibles de los padres y devuelve un nuevo diccionario con las probabilidades resultantes.

El código también incluye tres diccionarios, AA, Aa y aa, que representan los genotipos posibles de un individuo homocigoto dominante (AA), heterocigoto (Aa) y homocigoto recesivo (aa) respectivamente. Estos diccionarios se utilizan como puntos de partida para calcular las probabilidades de los genotipos en el árbol genealógico.

En la función principal main, el programa lee el árbol genealógico desde un archivo llamado "rosalind_mend.txt" y lo almacena en la variable pedigree. A continuación, procesa el árbol genealógico, reemplazando los caracteres especiales del formato Newick con la función replace(), y agrega la función cp en los lugares adecuados del árbol utilizando nuevamente replace().

Finalmente, el programa evalúa el árbol genealógico, tratándolo como una expresión de Python, utilizando eval(). De esta manera, se obtienen las probabilidades requeridas de los genotipos "AA", "Aa" y "aa" del individuo en la raíz del árbol.